EXHIBIT D

```
/* *********************************************************************
    File: ciphers.c

    SSL Plus: Security Integration Suite(tm)
    Version 1.1.1 -- August 11, 1997

    Copyright (c)1996, 1997 by Consensus Development Corporation
          Copyright (c)1997, 1998 by Aventail Corporation

    Portions of this software are based on SSLRef(tm) 3.0, which is
    Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
    was developed by Netscape Communications Corporation and Consensus
    Development Corporation.

    In order to obtain this software, your company must have signed
    either a PRODUCT EVALUATION LICENSE (a copy of which is included in
    the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
    licenses have different limitations regarding how you are allowed to
    use the software. Before retrieving (or using) this software, you
    *must* ascertain which of these licenses your company currently
    holds. Then, by retrieving (or using) this software you agree to
    abide by the particular terms of that license. If you do not agree
    to abide by the particular terms of that license, than you must
    immediately delete this software. If your company does not have a
    signed license of either kind, then you must either contact
    Consensus Development and execute a valid license before retrieving
    (or using) this software, or immediately delete this software.

    *********************************************************************

    File: ciphers.c    Data structures for handling supported ciphers

    Contains a table mapping cipherSuite values to the ciphers, MAC
    algorithms, key exchange procedures and so on that are used for that
    algorithm, in order of preference.

    ****************************************************************** */

#ifndef _CRYPTYPE_H_
#include <cryptype.h>
#endif

#ifndef _SSLCTX_H_
#include <sslctx.h>
#endif

#include <string.h>

extern SSLSymmetricCipher SSLCipherNull;
extern SSLSymmetricCipher SSLCipherDES_CBC;
extern SSLSymmetricCipher SSLCipherDES40_CBC;
extern SSLSymmetricCipher SSLCipherRC4_40;
extern SSLSymmetricCipher SSLCipherRC4_56;
extern SSLSymmetricCipher SSLCipherRC4_128;
extern SSLSymmetricCipher SSLCipher3DES_CBC;

/* Even if we don't support NULL_WITH_NULL_NULL for transport, we need a reference for startup
   */
SSLCipherSpec SSL_NULL_WITH_NULL_NULL_CipherSpec =
{   SSL_NULL_WITH_NULL_NULL,
    Exportable,
    SSL_NULL_auth,
    &SSLHashNullOpt,
    &SSLCipherNull
};

/* Disable non-exportable cipher suites to build an export only library */
#ifndef ENABLE_NONEXPORT_CIPHERS
#define ENABLE_NONEXPORT_CIPHERS 1
#endif

```

```
/* Disable exportable cipher suites to build a strong crypto only library */
#ifndef ENABLE_EXPORT_CIPHERS
#define ENABLE_EXPORT_CIPHERS 1
#endif

/* Reenable DH-anon only if you know you want to use Diffie-Hellman cipher suites:
    Enabling DH-anon leaves you open to a man-in-the-middle attack which can degrade your
     security to this level. */
#ifndef ENABLE_DH_ANON
#define ENABLE_DH_ANON 0
#endif

/* Reenable NULL encryption cipher suites only if you know for a fact you want to support
    unencrypted sessions. Unencrypted sessions do not provide data privacy and may be more
    vulnerable to attack than encrypted sessions. */
#ifndef ENABLE_NULL_CIPHERS
#define ENABLE_NULL_CIPHERS 0
#endif

#ifdef VIRGIN_SSLPLUS
/* Order by preference */
SSLCipherSpec KnownCipherSpecs[] =
{
#if ENABLE_NONEXPORT_CIPHERS
    {   SSL_RSA_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipher3DES_CBC
  },
    {   SSL_RSA_WITH_RC4_128_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherRC4_128 },
    {   SSL_RSA_WITH_RC4_128_MD5, NotExportable, SSL_RSA, &SSLHashMD5, &SSLCipherRC4_128 },
    {   SSL_RSA_WITH_DES_CBC_SHA, NotExportable, SSL_RSA, &SSLHashSHA1, &SSLCipherDES_CBC },
#endif
#if ENABLE_EXPORT_CIPHERS
    {   SSL_RSA_EXPORT_WITH_RC4_40_MD5, Exportable, SSL_RSA_EXPORT, &SSLHashMD5,
  &SSLCipherRC4_40 },
    {   SSL_RSA_EXPORT_WITH_DES40_CBC_SHA, Exportable, SSL_RSA_EXPORT, &SSLHashSHA1,
  &SSLCipherDES40_CBC },
#endif
#if ENABLE_DH_ANON && ENABLE_NONEXPORT_CIPHERS
    {   SSL_DH_anon_WITH_3DES_EDE_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
  &SSLCipher3DES_CBC },
    {   SSL_DH_anon_WITH_RC4_128_MD5, NotExportable, SSL_DH_anon, &SSLHashMD5,
  &SSLCipherRC4_128 },
    {   SSL_DH_anon_WITH_DES_CBC_SHA, NotExportable, SSL_DH_anon, &SSLHashSHA1,
  &SSLCipherDES_CBC },
#endif
#if ENABLE_NULL_CIPHERS && ENABLE_EXPORT_CIPHERS
    {   SSL_RSA_WITH_NULL_SHA, Exportable, SSL_RSA, &SSLHashSHA1, &SSLCipherNull },
    {   SSL_RSA_WITH_NULL_MD5, Exportable, SSL_RSA, &SSLHashMD5, &SSLCipherNull }
#endif
};

int CipherSpecCount = sizeof(KnownCipherSpecs) / sizeof(SSLCipherSpec);
#endif /* VIRGIN_SSLPLUS */

SSLErr
FindCipherSpec(SSLContext *ctx, uint16 specID, SSLCipherSpec* *spec)
{
  int i;
  uint32 mask;

    *spec = 0;
    for (i = 0; i < CipherSpecCount; i++)
    {
          if (KnownCipherSpecs[i].cipherSpec == specID)
        {
                  mask = (uint32) 1;
                  mask <<= i;
                  if(ctx->cipherspecs & mask)
                  {
                          *spec = &KnownCipherSpecs[i];
                          break;
                  }
        }
    }

    if (*spec == 0)         /* Not found */
        return SSLNegotiationErr;
    return SSLNoErr;
}

SSLErr SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
SSLErr SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
SSLErr SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
SSLErr SSLDESFinish(void *cipherRef, SSLContext *ctx);
SSLErr SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
SSLErr SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);

SSLSymmetricCipher SSLCipherDES_CBC = {
    8,      /* Key size in bytes */
    8,      /* Secret key size = 64 bits */
    8,      /* IV size */
    8,      /* Block size */
    SSLDESInit,
    SSLDESEncrypt,
    SSLDESDecrypt,
    SSLDESFinish,
  SSLDESExport,
  SSLDESImport
};

SSLSymmetricCipher SSLCipherDES40_CBC = {
    8,      /* Key size in bytes */
    5,      /* Secret key size = 40 bits */
    8,      /* IV size */
    8,      /* Block size */
    SSLDESInit,
    SSLDESEncrypt,
    SSLDESDecrypt,
    SSLDESFinish
};

typedef struct _DESState
{
  unsigned char key[24]; /* work for 3DES and DES both */
  unsigned char iv[8];
  int reading; /* do we really need this? */
  B_ALGORITHM_OBJ des;
} DESState;

SSLErr
SSLDESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
{
  SSLBuffer                     desState;
    B_ALGORITHM_OBJ              *des;
    static B_ALGORITHM_METHOD    *chooser[] = { &AM_DES_CBC_ENCRYPT, &AM_DES_CBC_DECRYPT, 0 };
    B_KEY_OBJ                    desKey;
    ITEM                         keyData;
    SSLErr                       err;
    int                          rsaErr;
  DESState *s;

    if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
        return err;
    s = (DESState *)desState.data;

  memcpy(s->key, key, 8);
  memcpy(s->iv, iv, 8);

    if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
        return SSLUnknownErr;
    if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, iv)) != 0)
        return SSLUnknownErr;
    if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
        return SSLUnknownErr;
    keyData.data = key;
    keyData.len = 8;
    if ((rsaErr = B_SetKeyInfo(desKey, KI_DES8, key)) != 0)
    {   B_DestroyKeyObject(&desKey);
        return SSLUnknownErr;
    }
    if (cipherRef == (void**)&(ctx->writePending.symCipherState))
    {
           s->reading = 0;
           if ((rsaErr = B_EncryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
        {
                    B_DestroyKeyObject(&desKey);
            return SSLUnknownErr;
        }
    }
    else if (cipherRef == (void**)&(ctx->readPending.symCipherState))
    {
           s->reading = 1;
           if ((rsaErr = B_DecryptInit(*des, desKey, chooser, &ctx->sysCtx.yield)) != 0)
        {
                    B_DestroyKeyObject(&desKey);
            return SSLUnknownErr;
        }
    }
    else
        ASSERTMSG("Couldn't determine read/writeness");

    B_DestroyKeyObject(&desKey);
    *cipherRef = (void*)s;
    return SSLNoErr;
}

SSLErr
SSLDESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
{
   DESState *s = (DESState *) cipherRef;
   void *subCipherRef = NULL;
     int                rsaErr;
     unsigned int           outputLen;
     SSLBuffer              temp;
     SSLErr             err;

   if(cipherRef == NULL)
           return SSLUnknownErr;

   if(iv != NULL)
   {
           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8,
                                                                 (POINTER) iv->data)) !=
   SSLNoErr)
                   return err;
   }
   else
   {
           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
                   return err;
   }

     ASSERT(src.length == dest.length);
     ASSERT(src.length % 8 == 0);

     if (src.data == dest.data)
/* BSAFE won't let you encrypt in place */
     {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
             return err;
         memcpy(temp.data, src.data, (size_t) src.length);
     }
    else
        temp = src;

    if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
                        (unsigned int) dest.length, temp.data,
                        (unsigned int) temp.length,
                    (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
    {   if (src.data == dest.data)
            SSLFreeBuffer(&temp, &ctx->sysCtx);
        return SSLUnknownErr;
    }

    ASSERT(outputLen == src.length);

    if (src.data == dest.data)
        SSLFreeBuffer(&temp, &ctx->sysCtx);

    if (outputLen != src.length)
        return SSLUnknownErr;

  /* if not doing SSLoppy, save the IV for next time... */
  if(iv == NULL)
  {
        unsigned char *buf;

        if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
                                                                        AI_DES_CBC_IV8))
            != SSLNoErr)
                return err;

        memcpy(s->iv, buf, sizeof(s->iv));
  }

/* memcpy(s->iv, dest.data + dest.length - 8, 8); */

    return SSLNoErr;
}

SSLErr
SSLDESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
{
   DESState *s = (DESState *) cipherRef;
    int              rsaErr;
    unsigned int        outputLen;
    SSLBuffer           temp;
    SSLErr          err;

   if(cipherRef == NULL)
         return SSLUnknownErr;

   if(iv != NULL)
   {
         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, (POINTER) iv->data))
            != SSLNoErr)
                return err;
   }
   else
   {
         if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_CBC_IV8, s->iv)) != SSLNoErr)
                return err;
   }

    ASSERT(src.length == dest.length);
    ASSERT(src.length % 8 == 0);

/* memcpy(s->iv, src.data + src.length - 8, 8); */

    if (src.data == dest.data)
/* BSAFE won't let you encrypt in place */
    {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
            return err;
        memcpy(temp.data, src.data, (size_t) src.length);
    }
    else
        temp = src;

    if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
                            (unsigned int) dest.length, temp.data,
                            (unsigned int) temp.length,
                        (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
    {   if (src.data == dest.data)
            SSLFreeBuffer(&temp, &ctx->sysCtx);
        return SSLUnknownErr;
    }

    ASSERT(outputLen == src.length);

    if (src.data == dest.data)
        SSLFreeBuffer(&temp, &ctx->sysCtx);

    if (outputLen != src.length)
        return SSLUnknownErr;

   /* if not doing SSLoppy, save the IV for next time... */
   if(iv == NULL)
   {
          unsigned char *buf;

          if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
                                                                      AI_DES_CBC_IV8))
             != SSLNoErr)
                return err;
          memcpy(s->iv, buf, sizeof(s->iv));
   }

    return SSLNoErr;
}

SSLErr
SSLDESFinish(void *cipherRef, SSLContext *ctx)
{
   DESState *s = (DESState *) cipherRef;
    SSLBuffer           desState;
    SSLErr          err;

   if(cipherRef == NULL)
          return SSLUnknownErr;

    B_DestroyAlgorithmObject(&(s->des));

   memset(cipherRef, 0, sizeof(DESState));
   desState.data = (unsigned char*)cipherRef;
    desState.length = sizeof(DESState);

    err = SSLFreeBuffer(&desState, &ctx->sysCtx);
    return err;
}

SSLErr SSLDESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
{
   DESState *s = (DESState *) cipherRef;

   if(cipherRef == NULL)
          return SSLUnknownErr;

   if(blob->length < (8 + 8))
          return SSLMemoryErr;

   memcpy(blob->data, s->key, 8);
   memcpy(blob->data + 8, s->iv, 8);
/* memcpy(blob->data + 16, &(s->reading), sizeof(int)); */
   blob->length = 16;
```

```

  return SSLNoErr;
}

SSLErr SSLDESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
{
  unsigned char *key, *iv;

  if(blob == NULL)
         return SSLUnknownErr;
  if(blob->length < 16)
         return SSLUnknownErr;

  key = blob->data;
  iv = blob->data + 8;

  return SSLDESInit(key, iv, cipherRef, ctx);
}


SSLErr SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx);
SSLErr SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
SSLErr SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext
    *ctx);
SSLErr SSL3DESFinish(void *cipherRef, SSLContext *ctx);
SSLErr SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob);
SSLErr SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob);

SSLSymmetricCipher SSLCipher3DES_CBC = {
    24,     /* Key size in bytes */
    24,     /* Secret key size = 192 bits */
    8,      /* IV size */
    8,      /* Block size */
    SSL3DESInit,
    SSL3DESEncrypt,
    SSL3DESDecrypt,
    SSL3DESFinish,
  SSL3DESExport,
  SSL3DESImport
};

SSLErr
SSL3DESInit(uint8 *key, uint8* iv, void **cipherRef, SSLContext *ctx)
{
  SSLBuffer                   desState;
  DESState *s;
   static B_ALGORITHM_METHOD   *chooser[] = { &AM_DES_EDE3_CBC_ENCRYPT,

  &AM_DES_EDE3_CBC_DECRYPT, 0 };
   B_KEY_OBJ                   desKey;
   ITEM                        keyData;
   SSLErr                      err;
   int                         rsaErr;

   if ((err = SSLAllocBuffer(&desState, sizeof(DESState), &ctx->sysCtx)) != 0)
       return err;
   s = (DESState *)desState.data;
   if ((rsaErr = B_CreateAlgorithmObject(&(s->des))) != 0)
       return SSLUnknownErr;
   if ((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, iv)) != 0)
       return SSLUnknownErr;
  memcpy(s->iv, iv, 8);

   if ((rsaErr = B_CreateKeyObject(&desKey)) != 0)
       return SSLUnknownErr;
   keyData.data = key;
   keyData.len = 24;
   if ((rsaErr = B_SetKeyInfo(desKey, KI_24Byte, key)) != 0)
   {
         B_DestroyKeyObject(&desKey);
```

```
        return SSLUnknownErr;
    }
   memcpy(s->key, key, 24);

    if (cipherRef == (void**)&(ctx->writePending.symCipherState))
    {
         if ((rsaErr = B_EncryptInit(s->des, desKey, chooser,
                                                     &ctx->sysCtx.yield)) != 0)
        {
                 B_DestroyKeyObject(&desKey);
            return SSLUnknownErr;
        }
    }
    else if (cipherRef == (void**)&(ctx->readPending.symCipherState))
    {
         if ((rsaErr = B_DecryptInit(s->des, desKey, chooser,
                                                     &ctx->sysCtx.yield)) != 0)
        {
                 B_DestroyKeyObject(&desKey);
            return SSLUnknownErr;
        }
    }
    else
        ASSERTMSG("Couldn't determine read/writeness");

    B_DestroyKeyObject(&desKey);
    *cipherRef = (void*)desState.data;
    return SSLNoErr;
}

SSLErr
SSL3DESEncrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
{
   DESState *s =(DESState *) cipherRef;
    int            rsaErr;
    unsigned int   outputLen;
    SSLBuffer      temp;
    SSLErr         err;

    ASSERT(src.length == dest.length);
    ASSERT(src.length % 8 == 0);
   if(cipherRef == NULL)
          return SSLUnknownErr;

   if(iv != NULL)
   {
          if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
                                                          (POINTER) iv->data)) !=
   SSLNoErr)
                 return err;
   }
   else
   {
          if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
                 return err;
   }


   if (src.data == dest.data)
/* BSAFE won't let you encrypt in place */
    {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
           return err;
        memcpy(temp.data, src.data, (size_t) src.length);
    }
    else
        temp = src;

    if ((rsaErr = B_EncryptUpdate(s->des, dest.data, &outputLen,
                        (unsigned int) dest.length, temp.data,
                        (unsigned int) temp.length,
                    (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
    {   if (src.data == dest.data)
            SSLFreeBuffer(&temp, &ctx->sysCtx);
        return SSLUnknownErr;
    }

    ASSERT(outputLen == src.length);

    if (src.data == dest.data)
        SSLFreeBuffer(&temp, &ctx->sysCtx);

    if (outputLen != src.length)
        return SSLUnknownErr;

   if(iv == NULL)
   {
           unsigned char *buf;

           if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
                                                                        AI_DES_EDE3_CBC_IV8))
              != SSLNoErr)
                  return err;
           memcpy(s->iv, buf, sizeof(s->iv));
   }

/*memcpy(s->iv, dest.data + dest.length - 8, 8); */

    return SSLNoErr;
}

SSLErr
SSL3DESDecrypt(SSLBuffer src, SSLBuffer dest, SSLBuffer *iv, void *cipherRef, SSLContext *ctx)
{
   DESState *s = (DESState *) cipherRef;
    int             rsaErr;
    unsigned int        outputLen;
    SSLBuffer           temp;
    SSLErr          err;

    ASSERT(src.length == dest.length);
    ASSERT(src.length % 8 == 0);
   if(cipherRef == NULL)
           return SSLNoErr;

   if(iv != NULL)
   {
           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8,
                                                                        (POINTER) iv->data)) !=
   SSLNoErr)
                   return err;
   }
   else
   {
           if((rsaErr = B_SetAlgorithmInfo(s->des, AI_DES_EDE3_CBC_IV8, s->iv)) != SSLNoErr)
                   return err;
   }

/* memcpy(s->iv, src.data + src.length - 8, 8);  */

    if (src.data == dest.data)
/* BSAFE won't let you encrypt in place */
    {   if (ERR(err = SSLAllocBuffer(&temp, src.length, &ctx->sysCtx)) != 0)
            return err;
        memcpy(temp.data, src.data, (size_t) src.length);
    }
    else
        temp = src;

    if ((rsaErr = B_DecryptUpdate(s->des, dest.data, &outputLen,
                          (unsigned int) dest.length, temp.data,
                          (unsigned int) temp.length,
                   (B_ALGORITHM_OBJ) 0, &ctx->sysCtx.yield)) != 0)
    {   if (src.data == dest.data)
            SSLFreeBuffer(&temp, &ctx->sysCtx);
        return SSLUnknownErr;
    }

  if(iv == NULL)
  {
        unsigned char *buf;

        if((rsaErr = B_GetAlgorithmInfo((POINTER *) &buf, s->des,
                                                                    AI_DES_EDE3_CBC_IV8)) !=
  SSLNoErr)
                return err;
        memcpy(s->iv, buf, sizeof(s->iv));
  }

    ASSERT(outputLen == src.length);

    if (src.data == dest.data)
        SSLFreeBuffer(&temp, &ctx->sysCtx);

    if (outputLen != src.length)
        return SSLUnknownErr;

    return SSLNoErr;
}

SSLErr
SSL3DESFinish(void *cipherRef, SSLContext *ctx)
{
  DESState *s = (DESState *) cipherRef;
    SSLBuffer           desState;
    SSLErr          err;

  if(cipherRef == NULL)
        return SSLUnknownErr;

  B_DestroyAlgorithmObject(&(s->des));

  memset(cipherRef, 0, sizeof(DESState));
    desState.data = (unsigned char*)cipherRef;
    desState.length = sizeof(DESState);
    err = SSLFreeBuffer(&desState, &ctx->sysCtx);
    return err;
}

SSLErr SSL3DESExport(void *cipherRef, SSLContext *ctx, SSLBuffer *blob)
{
  DESState *s = (DESState *) cipherRef;

  if(cipherRef == NULL)
        return SSLUnknownErr;

  if(blob->length < (24 + 8))
        return SSLMemoryErr;

  memcpy(blob->data, s->key, 24);
  memcpy(blob->data + 24, s->iv, 8);
  blob->length = 32;

  return SSLNoErr;
}

SSLErr SSL3DESImport(void **cipherRef, SSLContext *ctx, SSLBuffer *blob)
{
  unsigned char *key, *iv;

  if(blob == NULL)
        return SSLUnknownErr;
  if(blob->length < 32)
        return SSLUnknownErr;

    key = blob->data;
    iv = blob->data + 24;

    return SSL3DESInit(key, iv, cipherRef, ctx);
}
```